Guile Programmer's Manual

For use with Cygnus Guile 1.0
Last updated 6 July 1996

Mark Galassi
Los Alamos National Laboratory and Cygnus Support
rosalia@nis.lanl.gov

Copyright © 1996 Cygnus Support

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by Free Software Foundation.

1 What goes in this manual

You might be wondering why there are two separate manuals for Guile.

It is customary to split the documentation for major packages into a *user manual* (a gentle and introductory document) and a *reference manual*. Sometimes people go a step farther and make a separate *tutorial*; other times the tutorial is part of the user manual.

In this framekwork, what you are supposed to do is: use the user manual until you have understood all that it has to offer you, and then use the reference manual for the rest of your life (except when you are teaching).

This **Guile Programmer's Manual** is indeed a *reference manual*, so I assume that you know everything that's in the **Guile User Manual**, and you are using this manual to look up the exact detailed specification of what a procedure does, or what a variable means.

1.1 What does NOT go in this manual

You will find that this Programmer's Manual is mostly filled with detailed reference on what each procedure that Guile adds to scheme. More precisely: in a reductionist view, Guile could be viewed as a sum of its parts:

```
guile   =       standard scheme (R4RS)
        PLUS    extensions to R4RS offered by SCM
        PLUS    some extra primitives offered by Guile
        PLUS    portable scheme library (SLIB)
        PLUS    embeddable scheme interpreter library (libguile)
        PLUS    Tk toolkit
        PLUS    threads
        PLUS    Posix library (goonix)
        PLUS    OpenGL library (mesa)
        PLUS    OpenGL toolkit (glut)
        PLUS    Regular expression library (rx)
        PLUS    Applet formalism
        PLUS    Tcl library
```

So a complete reference on Guile would have to describe the tools given to the programmer by *each of these aspects of Guile!*

Fortunately many of these things have already been documented elsewhere, so we will skip them; some (like Tk) are described elsewhere, but there are language-related subtelties when they are used with Guile, so we document Tk here.

We will completely leave out the documentation of standard scheme, SLIB, mesa, glut and Tcl. You can find many references to this material in section "Where to find more Guile/scheme resources" in *Guile User Manual*.

There is no bibliography in the Programmer's Manual because the User Manual covers that (see section "Where to find more Guile/scheme resources" in *Guile User Manual*).

2 Guile Tk reference

2.1 Tk preliminaries

Some aspects of the Tk implementation for Guile are not yet in final form, so I will give you the recipe for using Tk in the *guile-r0.3* release. Remember to check this section for changes in new releases of Guile.

This chapter is written assuming that you are running the *guile* program directly. If you are writing a C program instead, and linking with *libguile*, you can follow the instructions in Chapter 6 [Libguile reference], page 19 to initialize Tk correctly, and then follow this chapter when you write scheme code.

The following lines should be at the top of your scheme source file:

```
(require 'Gwish)

(use-library tcl)
(use-interface tcl)
(use-interface tclhack)
```

Your program will then typically create widgets and bind callbacks to the widgets. To then enter the Tk main loop, you need to call:

```
(tk-main-loop)
```

It is important to remember that Tk widget callbacks must always be specified via the special Guile primitive `tcl-lambda`, rather than the ordinary scheme `lambda`. This is for two reasons: first, procedures created by `tcl-lambda` know that they are still referenced as callbacks, and thus don't get garbage collected away. Second, Guile's Tk implementation is still based on Tcl, and `tcl-lambda` knows how to convert Tcl calling syntax to scheme.

This is expected to change as: (1) Sun works on making Tk independent of Tcl, and (2) the Free Software Foundation makes Guile independent of Tcl.

2.2 Creating widgets

The Tk package adds several primitive procedures to scheme; most of these correspond to Tk widget classes, and when invoked they create a new widget. Most of these procedures take the same basic set of options, but then each widget class has its own specific options.

Each widget type is described in detail in the very complete *Tk man pages*, so I will only give a brief description of them here.

The User Manual (see section "Creating a widget" in *Guile User Manual*) got you used to going back and forth between the Tcl/Tk and Guile/Tk calling conventions. The thing to remember here is that `pathName` arguments are always scheme quoted symbols, and the variables begin with a . (period).

button *pathName* [*options* ...] Tk Widget Creation
canvas *pathName* [*options* ...] Tk Widget Creation
checkbutton *pathName* [*options* ...] Tk Widget Creation
entry *pathName* [*options* ...] Tk Widget Creation
frame *pathName* [*options* ...] Tk Widget Creation
label *pathName* [*options* ...] Tk Widget Creation
listbox *pathName* [*options* ...] Tk Widget Creation
menu *pathName* [*options* ...] Tk Widget Creation
menubutton *pathName* [*options* ...] Tk Widget Creation
message *pathName* [*options* ...] Tk Widget Creation
listbox *pathName* [*options* ...] Tk Widget Creation
radiobutton *pathName* [*options* ...] Tk Widget Creation
scale *pathName* [*options* ...] Tk Widget Creation
scrollbar *pathName* [*options* ...] Tk Widget Creation
text *pathName* [*options* ...] Tk Widget Creation
toplevel *pathName* [*options* ...] Tk Widget Creation
update *pathName* [*options* ...] Tk Widget Creation

These procedures create new widgets of the respective type. The *options* are described in the very detailed man pages for each widget, but also in the special `options` man page provided by Tk.

tk_popup *menu x y* [*entries* ...] Tk Widget Creation

Posts a popup menu at location *(x, y)* on the screen. The *menu* widget should already have been created. The *entry* parameter is the index in the menu to which the mouse will point by default.

Widgets are destroyed with the `destroy` procedure:

destroy [*windows* ...] Tk Widget Creation

Deletes the given *windows* plus all their descendants. If the root window `"."` is destroyed, then the entire application will be destroyed.

2.3 Widgets as procedures

Once you have created a widget, it automatically becomes a new scheme procedure. This procedure can be invoked with arguments, and such invocation will configure or modify the widget.

WIDGET [*options* ...] Tk Widget Configuration

Configure WIDGET according to the *options*. Once again: *options* are described in detail in the *options* man page.

2.4 Geometry management procedures

The basic procedure used by Tk to arrange widgets on the screen is the packer. To talk to it, use:

pack *option arg* [*args* ...] Tk Geometry Management

This geometry manager packas the child windows in order around the edges of the parent.

Tk is shipped with two other geometry management mechanisms: `place`, and a way of arranging objects in canvases.

place *window option value* [{*option value*} ... Tk Geometry Management
place *configure window option value* [{*option value*} ...] Tk Geometry Management
forget *window* Tk Geometry Management
info *window* Tk Geometry Management
slaves *window* Tk Geometry Management

This geometry manager fixed placement of windows, where you specify the exact size and location of the master and slave. This is seldom used, since `pack` is the more common Tk geometry manager.

Tk widgets also interact properly with the window manager (as long as you have a good window manager). There is a procedure which allows you to explicitly talk to the window manager:

wm *option window* [*args* ...] Tk Geometry Management

Interacts with the window manager, allowing the Tk programmer to set such resources as the title, geometry, resize increments, ... of that window. There are many possible values for *option*, documented in the Tk `wm` man page.

2.5 Event handling procedures

Whenever an action happens on the display that might be interesting to the Guile/Tk application (mouse motion, mouse clicks, keyboard events), Tk generates an *event*. The application can install event handlers to be invoked when these events occur.

Many of the event handlers are *widget callbacks*, which are treated in Section 2.2 [Creating widgets], page 2; here we discuss other procedures that relate to event handling and flow control in Tk.

bind *widget* "<*BUTTON-ACTION*>" *callback* Tk Event Handling

Binds the given button action to the procedure *callback*.

fileevent *input-port 'readable* [*callback*] Tk Event Handling
fileevent *output-port 'writeable* [*callback*] Tk Event Handling

Creates an event handler for the open file associated with *input-port* or *output-port*. The event handler invokes the given *callback* procedure when the file becomes available for reading (or writing, respectively).

focus Tk Event Handling
focus *window* Tk Event Handling
focus *option* [*args* ...] Tk Event Handling

Manages the input focus. Without arguments it returns the path name of the focus window. With a single *window* argument it sets the focus to that window. See the `focus` man page for a description of the third form.

grab [*:global*] *window* Tk Event Handling
grab *option* [*args* ...] Tk Event Handling

Confine pointer and keyboard events to the subtree of the specified *window*. The second form has several behaviours according to the value of *option*, which can be `current`, `release`, `set`, `status`.

tk-main-loop Tk Event Handling

Enters the Tk loop, which waits for events and invokes the appropriate callbacks. This is typically called after all widgets have been set up. It never returns until the . window is destroyed. This means that after invoking `(tk-main-loop)` in a Guile/Tk program, all future computation has to be done in widget callbacks.

tkwait *variable name* Tk Event Handling
tkwait *visibility name* Tk Event Handling
tkwait *window name* Tk Event Handling

Wait for a variable to change, or a window to be destroyed.

update [*'idletasks*] Tk Event Handling

Process pending events, flush all output, process *when-idle* handlers, until there are no pending events. If the *'idletasks* symbol is passed, only process *when-idle* handlers.

2.6 Miscellaneous Tk procedures

Here are some procedures that come from the Tk module, and which do not belong in the above sections. As usual, complete details are available in the man pages.

after *ms* Tk Misc
after *ms* [*commands ...*] Tk Misc
after *cancel id* Tk Misc
after *cancel* [*commands ...*] Tk Misc
after *idle* [*commands ...*] Tk Misc

Exececute the given *commands* after *ms* microseconds.

bell [*display*] Tk Misc

Rings the bell on the *display*, or the default display if no *display* argument is given.

bindtags *window* [*tagList*] Tk Misc

Returns a string describing the bindings associated with the given window.

clipboard *cmd* [*args ...*] Tk Misc

Allows manipulation of the Tk clipboard. The *cmd* argument selects what type of action is taken on the clipboard. This can be `clear` or `append`.

exit [*returnCode*] Tk Misc

Exits the process. If *returnCode* is not given it defaults to 0.

image *option* [*args ...*] Tk Misc

The image command can create and manipulate images. The *option* parameter can be `create`, `delete`, `height`, `names`, `type`, `width`. The builtin image types are *bitmap* (for black and white images) and *photo* for full-color pixmaps.

lower *window* [*belowThis*] Tk Misc

Without arguments, `lower` will place the given *window* under all its siblings in the stacking order. If *belowThis* is a window pathname, then *window* will be placed right underneath *belowThis*.

option *add pattern value* [*priority*] Tk Misc
option *clear* Tk Misc
option *get window name class* Tk Misc
option *readfile fileName* [*priority*] Tk Misc
Add (or retrieve) window options to (or from) the option database.

raise *window* [*aboveThis*] Tk Misc
Without arguments, `raise` will place the given *window* over all its siblings in the stacking order. If *aboveThis* is a window pathname, then *window* will be placed right above *aboveThis*.

selection *option* [*args* ...] Tk Misc
Allows manipulation of the X selection.

send [*options* ...] *app cmd* [*args* ...] Tk Misc
Arranges for a command *cmd* to be executed in the application *app*.

tk *option* [*args* ...] Tk Misc
Manipulates internal state variables in Tk.

winfo *option* [*args* ...] Tk Misc
Retrieve information about Tk's windows. The *option* argument specifies what type of information; it can be `'atom`, `'atomname`, `'cells`, `'children`, `'class`, `'colormapfull`, `'containing`, `'depth`, `'exists`, `'fpixels`, `'geometry`, `'height`, `'id`, `'interps`, `'ismapped`, `'manager`, `'name`, `'parent`, `'pathname`, `'pixels`, `'pointerx`, `'pointerxy`, `'pointery` and many others listed in the `winfo` man page.

3 Goonix reference

3.1 Goonix preliminaries

Goonix is a *posix-compliant* (*John S. Quarterman and Susanne Willhelm: "UNIX, POSIX, and Open Systems", Addison-Wesley Publishing Co, Inc. 1993*) library for systems programming in scheme. It allows the scheme programmer to use the same calls available to C programmers in a posix environment.

This chapter is written assuming that you are running the *guile* program directly. If you are writing a C program instead, and linking with *libguile*, you can follow the instructions in Chapter 6 [Libguile reference], page 19 to initialize goonix correctly, and then follow this section when you write scheme code.

Unlike the Tk package, no special scheme code is required to initialize goonix: all goonix procedures are primitive, and available as long as goonix has been initialized with the correct libguile calls.

When documenting the procedures in this chapter, I use the convention *[arg]* to indicate that the argument is optional, and *[args ...]* to indicate that the optional argument can be repeated.

You will notice that most of these procedures closely mirror the equivalent UNIX system or library calls. If you want to know more detail about how these procedures behave, you can read the relevant UNIX man page.

You will also notice that many of these procedures begin with a `%` character. There is a convention in goonix that puts a `%` in front of all procedures that might invoke the scheme `(error ...)` mechanism, thus escaping from the current continuation.

3.2 I/O extensions to standard scheme

These *I/O Extensions* to scheme provide many of the UNIX file system calls, thus allowing sophisticated file manipulation and directory traversal. This is mostly done using UNIX's `stdio` layer, rather than the raw `open`, `read`, `write` layer.

Standard I/O *file pointers* are directly translated to scheme *ports*. Ports are described in great detail in all the scheme references (see section "Where to find more Guile/scheme resources" in *Guile User Manual*), so I will simply remind you that an input port is a scheme object which can deliver characters, and an output port is a scheme object which can accept characters. It is OK to think of ports as file pointers, even though there are some subtelties in ports related to scheme's unique control structure.

In the following read/write routines, if *port* is omitted, standard input (or standard output) is assumed.

read-line [*port*] I/O Extensions

Reads a line of text from the given input port, and returns the string. If *port* is not specified, standard input is assumed.

read-line! *str* [*port*] I/O Extensions

A destructive version of the above: puts the string in *str*; return value is unspecified.

write-line *obj* [*port*] I/O Extensions
Writes a representation of *obj* to *port*. If port is not specified, it is assumed to be standard output.

%ftell *port* I/O Extensions
Returns the current seek pointer of the given port.

%fseek *port offset whence* I/O Extensions
Sets the seek pointer for *port* to the position *offset*. If *whence* is 0, that offset is with respect to the beginning of the file. If *whence* is 1, the offset is with respect to the current position. If *whence* is 2, the offset is with respect to the end of the file.

%freopen *filename modes port* I/O Extensions
Opens *filename* with the specified *modes*, and associates *port* with that file.

%duplicate-port *old-port modes* I/O Extensions
Returns a new port having the same open file as *old-port*.

%redirect-port *into-port from-port* I/O Extensions
Redirects I/O from *from-port* to *into-port*: from here on, *into-port* will refer to the same file as *from-port*. If *into-port* already refers to an open file, it will be closed.

%opendir *dirname* I/O Extensions
Opens the directory *dirname*, returning a directory port which can be used with `%readdir` and `closedir`.

%readdir *dirp* I/O Extensions
Gets the next file name from the directory port *dirp*.

rewinddir *dirp* I/O Extensions
Repositions the directory port *dirp* to the beginning of the stream.

%closedir *dirp* I/O Extensions
Closes the directory port *dirp*.

%mkdir *dirname* [*mode*] I/O Extensions
Makes a new directory with name *dirname*. The directory permissions are given by the optional argument *mode*, or by the current *umask* if *mode* is not specified.

%rmdir *dirname* I/O Extensions
Removes the directory with path *dirname*.

%chdir *path* I/O Extensions
Changes the current working directory to *path*.

%getcwd I/O Extensions
Returns a string with the current working directory.

%chmod *filename mode* I/O Extensions
Sets the permission mode on *filename* to *mode*.

%utime *path* [*actime* [*modtime*]] I/O Extensions

Sets the access and modification times of the file at *path*. The time values *actime* and *modtime* are given in standard UNIX time: the number of seconds since 00:00:00 GMT, Jan. 1 1970. If *actime* and *modtime* are not given, the current time is used.

umask [*mask-value*] I/O Extensions

Sets the user's `umask` to be the specified value *mask-value*. The new setting is returned. If no *mask-value* is given, `umask` will return the current setting.

Notice that the UNIX `umask` command interprets the *mask-value* to be an octal integer, whereas the goonix `umask` call does not make any such conversion, so you have to explicitly tell scheme to use an octal radix, or think about permissions in decimal!

%rename *old-fname new-fname* I/O Extensions

Renames the file from *old-fname* to *new-fname*. This is analogous to the UNIX `mv` command and the `rename` system call.

%fileno *port* I/O Extensions

Returns the integer file descriptor associated with *port*. In case of error, `#f` is returned.

%isatty *port* I/O Extensions

Returns `#t` if the port is associated with a terminal device, and `#f` otherwise.

%fdopen *fd type* I/O Extensions

Returns an I/O port associated with the numeric file descriptor *fd* (which is assumed to be already open). The *type* argument is a standard I/O character string specifying whether the file should be opened for reading, writing, or appending:

```
r         open for reading

w         truncate or create for writing

a         append: open for writing at end of  file,  or
          create for writing

r+        open for update (reading and writing)

w+        truncate or create for update

a+        append; open or create for update at EOF
```

%primitive-move->fdes *port fd* I/O Extensions

Moves the file descriptor that underlies *port* to the given value *fd*.

Returns `#f` for error, 0 if the file descriptor is already equal to *fd*, and 1 if the file descriptor is successfully moved.

%access *path how* I/O Extensions

Determines the accessibility of the file in *path*. The *how* variable can take on any of the following values:

```
R_OK            test for read permission

W_OK            test for write permission

X_OK            test for execute or search permission

The following value may also be supplied for mode:

F_OK            test whether the directories leading  to
                the  file  can  be searched and the file
                exists.
```

%stat *filename* I/O Extensions

Gets typical UNIX *stat* information on *filename*, and returns that information in a scheme vector, where the UNIX `stat` structure values are stored in the following order: `#(st_dev st_ino st_mode st_nlink st_uid st_gid st_rdev st_size st_atime st_mtime st_ctime st_blksize st_blocks)`

getpid I/O Extensions

Returns the current process ID.

%putenv *str* I/O Extensions

Takes a string *str* of the form `"VARIABLE=value"` and adds that to the user's environment.

3.3 Posix system and library calls

%chown *path owner group* POSIX Calls

Changes the ownership of *path* to the specified *owner* and *group*.

%link *oldpath newpath* POSIX Calls

Adds *newpath* as a hard link to *oldpath*.

%pipe POSIX Calls

Creates a pipe, and returns a scheme pair with the *read* port in its `car` and the *write* port in its *cdr*. If the pipe creation fails, `%pipe` returns `#f`.

open-pipe *pipe-str mode* POSIX Calls

Opens a pipe to a new process by running *pipe-str*, and returns a port which can be used to write to the process (if *mode* is `"w"`) or read from it (if *mode* is `"r"`).

open-input-pipe *pipe-str* POSIX Calls

A shortcut for running `(open-pipe pipe-str "r")`.

open-output-pipe *pipe-str* POSIX Calls

A shortcut for running `(open-pipe pipe-str "w")`.

%getgroups POSIX Calls

Returns a vector with all the supplementary group IDs of the current user process.

%getpwuid [*name*] POSIX Calls

Given the login-name or uid in *name*, `%getpwuid` returns a vector with the following data: `#(login-name password uid gid GEGOS home-dir shell)`.

If *name* is not specified, the first entry in the password file is returned (or the *next* entry, upon successive invocations).

setpwent [*arg*] POSIX Calls

If called with an argument, `setpwent` rewinds the password file so the next call to `%getpwuid` will start from the beginning.

Without arguments, `setpwent` will close the password file; it can be used when processing is complete.

%getgrgid [*which-group*] POSIX Calls

If the argument *which-group* (either the group name or the gid) is given, `%getgrgid` returns the group file entry for that group. If *which-group* is not given, the first entry in the group file is returned (or the *next* entry, upon successive invocations).

setgrent [*arg*] POSIX Calls

This is analogous to `setpwent`, but for the group file.

If called with an argument, `setgrent` rewinds the group file so the next call to `%getgrgid` will start from the beginning.

Without arguments, `setgrent` will close the group file; it can be used when processing is complete.

%kill *pid sig* POSIX Calls

Sends a signal *sig* to the process *pid*. The possible values for *sig* are the usual UNIX signal types. Notice the order of arguments: this is the order of the UNIX `kill` system call, not the order of the `kill` command usually typed at the shell.

%waitpid *pid* [*options*] POSIX Calls

Waits for some or all child processes to exit or return a status.

If *pid* is -1, `%waitpid` will wait on any of its children. This is equivalent to the traditional UNIX `wait` system call.

If *pid* is greater than 0, `%waitpid` will wait on that particular child process.

If *pid* is equal to 0, `%waitpid` will wait on any children in its own process group.

If *pid* is less than -1, `%waitpid` will wait on any children in the process group of the particular child `abs(pid)`.

The *options* argument is a bitwise OR of any of the flags:

WNOHANG Tells `%waitpid` to not suspend if status is not immediately available for one of the relevant child processes.

WUNTRACED Also reports status of children that are stopped and have not yet been reported.

The value returned by `%waitpid` is a scheme pair of the child pid and the status returned for that child.

getppid POSIX Calls

Returns the process id of the current process's parent.

getuid POSIX Calls
geteuid POSIX Calls
getgid POSIX Calls
getegid POSIX Calls

Returns the real or effective user or group id of the current process.

%setuid *id* POSIX Calls
%seteuid *id* POSIX Calls
%setgid *id* POSIX Calls
%setegid *id* POSIX Calls

Sets the real or effective user or group id of the current process.

ttyname *port* POSIX Calls

Returns a string with the path of the terminal device associated with the *port*, or `#f` if the device is not a terminal. Notice how this also serves the function of the UNIX `isatty()` call.

%execl *command-str* [*exec-args* ...] POSIX Calls
%execlp *command-str* [*exec-args* ...] POSIX Calls

Similar to `execl`, except that if *filename* does not contain a slash it searches for the file in the directories listed in the `PATH` environment variable.

%execlp is similar, but returns `#f` if an error occurs.

%fork POSIX Calls

This is a raw interface to the UNIX `fork()` system call. It creates a new process with the same core image as the current process. `%fork` returns `#f` upon failure; otherwise the child gets a return value of 0, and the parent gets the child's pid.

See the UNIX `fork` man page for some of the subtelties of `%fork` (open file descriptors, directory streams, semaphores, *tms_* values, file locks, ...).

%select *reads writes excepts secs msecs* POSIX Calls

The `%select` call is used to examine a set of file descriptors (passed as scheme lists of non-negative integers in *readfds*, *writefds* and *exceptfds*).

The purpose of this examination is to determine if the descriptors are ready for the requested operation (reading or writing or treatment of exception). The return value is a list of 3 lists; the three lists are the original lists stripped down to only the *ready* file descriptors.

A timeout can be specified (in seconds plus miliseconds) with the *secs* and *msecs* are used.

%uname POSIX Calls

This is an interface to the UNIX `uname()` system call. Upon successful completion it returns a list of 5 strings describing the system name and OS version: `(list OSname nodename OSrelease OSversion MachineType)`. Upon failure it returns `#f`.

environ [*env-list*] POSIX Calls

With no argument, returns the current environment as a scheme list of `"VAR=val"` strings. It can be called with a list of `"VAR=val"` strings, and then the environment will be set to those values.

3.4 UNIX system and library calls

%mknod *path mode dev* UNIX Calls

Creates a new file named by the string *path*. The file permissions are specified in *mode*, and some special *file type* bits can also be OR-ed in (see the `mknod()` man page for details on the file type bits).

The *dev* argument is ignored unless *mode* is a block or character special file, in which case *dev* is a *configuration-dependent specification of a character or block I/O device*.

%acct *path* UNIX Calls

Turns on or off process accounting. The accounting information will be reported in the file named by the string *path*. If *path* is `#f`, accounting will be turned off.

Must be super user to use this call.

%nice *increment* UNIX Calls

Changes the *nice* value of a UNIX process by *increment* (remember: a large nice value means *less* priority). Only the super user can set a negative increment.

The value of *increment* should be between -20 and 20; if it is outside these bounds it will be silently adjusted to the extreme value.

sync UNIX Calls

Schedules a write to disk of all information in core memory that should be on disk, such as super blocks, inodes and buffered disk I/O.

%symlink *oldpath newpath* UNIX Calls

Makes *newpath* be a symbolic link to *oldpath*.

%readlink *path* UNIX Calls

Returns a string with the *contents of the symlink* at *path*. The *contents* is the path to the real file. Returs `#f` on any error condition.

%lstat *path* UNIX Calls

Like `%stat` except that if *path* is a symlink, `%lstat` will return information about the symlink, whereas `%stat` will look at the referenced file. The rationale behind this is that symlinks should be transparent to the traditional UNIX file system calls, but extra calls (like `%lstat` and `%readlink`) are provided to get more information.

4 Guile threads reference

Here is a the reference for Guile's threads. In this chapter I simply quote verbatim Tom Lord's description of the low-level primitives written in C (basically an interface to the POSIX threads library) and Anthony Green's description of the higher-level thread procedures written in scheme.

When using Guile threads, keep in mind that each guile thread is executed in a new dynamic root.

4.1 Low level thread primitives

with-new-thread *thunk error-thunk* Function

Evaluate *(thunk)* in a new thread, and new dynamic context, returning a new thread object representing the thread.

If an error occurs during evaluation, call error-thunk, passing it an error code describing the condition. [Error codes are currently meaningless integers. In the future, real values will be specified.] If this happens, the error-thunk is called outside the scope of the new root – it is called in the same dynamic context in which with-new-thread was evaluated, but not in the callers thread.

All the evaluation rules for dynamic roots apply to threads.

join-thread *thread* Function

Suspend execution of the calling thread until the target *thread* terminates, unless the target *thread* has already terminated.

yield Function

If one or more threads are waiting to execute, calling yield forces an immediate context switch to one of them. Otherwise, yield has no effect.

make-mutex Function

Create a new mutex object.

lock-mutex *mutex* Function

Lock *mutex*. If the mutex is already locked, the calling thread blocks until the mutex becomes available. The function returns when the calling thread owns the lock on *mutex*.

unlock-mutex *mutex* Function

Unlocks *mutex* if the calling thread owns the lock on *mutex*. Calling unlock-mutex on a mutex not owned by the current thread results in undefined behaviour. Once a mutex has been unlocked, one thread blocked on *mutex* is awakened and grabs the mutex lock.

4.2 Higher level thread procedures

with-new-thread *thunk error-thunk* Function

Evaluate *(thunk)* in a new thread, and new dynamic context, returning a new thread object representing the thread.

If an error occurs during evaluation, call error-thunk, passing it an error code describing the condition. [Error codes are currently meaningless integers. In the future, real values will be specified.] If this happens, the error-thunk is called outside the scope of the new root – it is called in the same dynamic context in which with-new-thread was evaluated, but not in the callers thread.

All the evaluation rules for dynamic roots apply to threads.

join-thread *thread* Function

Suspend execution of the calling thread until the target *thread* terminates, unless the target *thread* has already terminated.

yield Function

If one or more threads are waiting to execute, calling yield forces an immediate context switch to one of them. Otherwise, yield has no effect.

make-mutex Function

Create a new mutex object.

lock-mutex *mutex* Function

Lock *mutex*. If the mutex is already locked, the calling thread blocks until the mutex becomes available. The function returns when the calling thread owns the lock on *mutex*.

unlock-mutex *mutex* Function

Unlocks *mutex* if the calling thread owns the lock on *mutex*. Calling unlock-mutex on a mutex not owned by the current thread results in undefined behaviour. Once a mutex has been unlocked, one thread blocked on *mutex* is awakened and grabs the mutex lock.

5 Guile applet specification

This chapter has been written by Gordon Irlam, gordoni@cygnus.com, at Cygnus Support, February 28, 1996. It is included here almost verbatim.

This section provides the specification for writing Guile applets for use with the SurfIt! demo browser.

5.1 HTML applet extension

Guile Scheme applets are denoted by the mime type application/guile. This is the default type for the file extension .scm.

An applet can be inlined within a HTML document.

`<a href="applet.scm" rel=embed>`
`fallback_html`
`</a>`
: The applet applet.scm will be loaded, inserted into the page, and run when the document containing it is first loaded. fallback_html will be displayed by browsers that don't support Guile applets.

An applet may also be invoked when a hyperlink is followed.

`<a href="applet.scm"`
`fallback_html`
`</a>`
: When the hyperlink is selected the applet applet.scm will be loaded and executed. The original document will be cleared from the page only if the applet explicitly requests it to be. fallback_html will be displayed by browsers that don't support Guile applets.

5.2 Applet requirements

An applet is required to be a syntactically well formed Guile scheme program. When an applet is invoked the corresponding Guile program is retrieved, loaded, and executed.

Every applet is required to include the applet library:

`(require 'applet)`
: Incorporate the applet library into a Guile application. This is needed as a prerequisite to performing any of the other applet commands.

Every applet is required to define a routine to be called by the browser when the browser requires the applet to terminate execution:

define-applet-terminate *routine* Applet API
: *routine* is the name of a routine that will be invoked by the browser when the applet is required to terminate execution. If the applet is visible it should destroy the Tk canvas upon which it is visible and then exit.

An applet will persist for as long as it is accessible – either externally through it's being displayed on a page, or internally through the scheme environment.

5.3 Applet API

All applets reside in the same top level environment. This allows state to be shared between applets and to persist between applet invocations.

An applet has access to all the features of a regular Guile scheme program and to the gtcl/gtk Guile extensions. The features of gtk allow the Guile applet to interactively display itself within the parent document.

Note: While Guile provides the ability to control namespaces, and this is necessary to provide a secure environment within which applets can be run, this has not been done for the SurfIt! Guile demo browser. The SurfIt! Guile demo browser is mainly intended as a research prototype, not a production web browser.

browser-window-name Variable

browser-window *args* Applet API

browser-window-name is a string containing the Tk window name of the browser window. `browser-window` is the corresponding Tcl proc to which window commands can be sent.

applet-window-name Variable

applet-window *args* Appplet API

applet-window-name is a string containing a Tk window name for a frame that can be used to contain the applet. `applet-window` is the corresponding Tcl proc to which window commands can be sent. The frame still needs inserting into the browser window, assuming the applet is visible.

applet-embedindex Variable

applet-embedindex is a string containing the offset within the browser window of the applet's anchor. Most commonly this is used when insert the applet into the browser window at the same location as the original html:

```
(browser-window 'window 'create applet-embedindex :window
                applet-window-name).
```

applet-newpage Applet API

This routine clears the current contents of the browser window.

applet-parsehtml *html* Applet API

This routine parses the html specified by html and renders the result adding it to bottom of the browser window.

applet-loadurl *url* Applet API

This routine loads and renders the contents of the object specified by URL url inserting it into the applet window.

applet-loaddata *url* Applet API

This routine loads and returns the contents of the object specified by URL url. Note: This particular routine will probably not be present in guile-r0. Its suggested incorporation into the API, and it's syntax are still tentative.

5.4 Applet behavior

Applets should not monopolize the cpu. Instead they should be written using a callback or event loop polling based style so that the Tk event handler can continues to operate, and the user can continue to interact with the browser.

5.5 Browser behavior

Any run time error occurring within an applet that goes uncaught will cause the applet to exit, but the browser will continue to function.

6 Libguile reference

The Guile interpreter is essentially Aubrey Jaffer's *SCM* interpreter (see section "Overview" in *SCM: a portable scheme interpreter*) with some modifications to make it suitable as an embedded interpreter.

Part of the modification has been to provide a restricted interface to limit access to the SCM internals; this is called the `gscm_` interface, or *libguile* interface.

If you are *programming with Guile*, you should only use the C subroutines described in this manual, which all begin with `gscm_`.

If instead you are *extending Guile*, you have the entire SCM source to play with. This manual will not help you at all, but you can consult Aubrey Jaffer's SCM manual (see section "Internals" in *SCM: a portable scheme interpreter*).

If you are *adding a module to Guile*, I recommend that you stick to the `gscm_` interface: this interface is guaranteed to not change drastically, while the SCM internals might change as Guile is developed.

6.1 Libguile preliminaries

To use libguile, you must have the following toward the beginning of your C source:

```
#include <guile/gscm.h>
```

When you link, you will have to add at least `-lguile` to the list of libraries. If you are using more of Guile than the basic scheme interpreter, you will have to add more libraries.

- If you use the *Tk toolkit*, add `-lgtcltk -ltk4.1 -ltcl7.5 -lX11`.
- If you use the posix library *goonix*, add `-lgoonix`.
- If you use the *threads package*, add `-lthreads`.

If you use these extra packages, make sure you look at Section 6.3 [Starting and controlling the interpreter], page 20: it will show you what startup code you need to initialize each of these extra libraries.

6.2 Data types defined by libguile

The following C constants and data types are defined in libguile:

GSCM_status Data type

A data type returned by many `gscm_` routines. Its value is meant to be interpreted by `gscm_error_msg()` if it is not `GSCM_OK`.

GSCM_top_level Data type

The data type used to store information about the scheme top levels.

SCM Data type

This is a C data type used to store all scheme data, no matter what the scheme type. Values are converted between C data types and the SCM type with utility functions described below (see Section 6.7 [Converting data between C and scheme], page 22).

GSCM_OK Constant

A constant returned by `gscm_` calls when there was no error.

6.3 Starting and controlling the interpreter

In almost every case, your first `gscm_` call will be

`GSCM_status` **gscm_run_scm** (`int argc, char **argv, FILE stdin, FILE stdout, FILE stderr, (GSCM_status init_proc)(), int boh, char *boh`) Function

Starts up a scheme interpreter, passing *argc* and *argv*, with the given assignment of *stdin*, *stdout*, *stderr*. The routine `init_proc()` is invoked to initialize particular Guile packages.

This next batch of routines are the ones that can be included in the routine `init_proc()` passed to `gscm_run_scm`.

`void` **gscm_threads_init_all** () Function

Initializes Guile threads.

`void` **scm_init_unix** () Function

`void` **scm_init_posix** () Function

`void` **scm_init_ioext** () Function

These three initialize the *posix* library for scheme.

`void` **scm_init_gtcl** () Function

Initializes *tcl* support for Guile.

`void` **scm_init_gtk** () Function

Initializes *Tk* support for Guile.

After initializing the interpreter with `gscm_run_scm`, you need to create a top level. The *top level* variable you obtain will be used for most future libguile calls.

`GSCM_status` **gscm_create_top_level** (`GSCM_top_level *toplev`) Function

This routine creates a top level of the interpreter in which to evaluate scheme expressions.

At the end, when you are done with scheme, you can invoke:

`void` **gscm_destroy_top_level** (`GSCM_top_level toplev`) Function

This routine releases all the resources associated with that top level, thus allowing the top level to be garbage collected.

6.4 Error messages

If a routine returns a value of type `GSCM_status`, we can get a human-readable representation of what the error condition was by invoking:

`char *` **gscm_error_msg** (`GSCM_status status`) Function

This routine returns a string which can be printed directly. Note that the string will be trashed and reallocated with the next invocation of `gscm_error_msg`. Here's the typical example of the use of `GSCM_status`:

```
status = gscm_some_function_returning_status(...);
if (status != GSCM_OK) {
  fputs(gscm_error_msg(status), stderr);
  fputc('\n', stderr);
  exit(1);
}
```

Here is how the various possible error codes are defined in 'gscm.h':

```
#define GSCM_OK                         0
#define GSCM_QUIT                       1
#define GSCM_RESTART                    2
#define GSCM_ILLEGALLY_REENTERED        3
#define GSCM_OUT_OF_MEM                 4
#define GSCM_ERROR_OPENING_FILE         5
#define GSCM_ERROR_OPENING_INIT_FILE    6
```

6.5 Executing scheme code

Once you have an interpreter running, and you have created a top level environment, you can ask the interpreter to evaluate scheme code. There are two calls that implement this:

`GSCM_status` **gscm_eval_str** (`char **answer, GSCM_top_level toplev, char *scheme_code`) Function

This asks the interpreter to evaluate a single line of scheme code. The result of the evaluation is placed in the string **answer*. Note that answer is `malloc`-ed by `gscm_eval_str`, so after using the value of **answer*, you should free it. If *answer* is NULL, the evaluation result is not returned to the caller

Also note that the line of code in *scheme_code* must be a well formed scheme expression. If you have many lines of code you must either concatenate them into one string, or use `gscm_eval_file()`.

`GSCM_status` **gscm_eval_file** (`char **answer, GSCM_top_level toplev, char *fname`) Function

Completely analogous to `gscm_eval_str()`, except that a whole file is evaluated instead of a string.

6.6 Defining new scheme procedures in C

The real interface between C and scheme comes when you can write new scheme procedures in C. This is done through the routine

`void` **gscm_define_procedure** (`char *name, SCM (*fn)(), int req, int opt, int varp, char *doc`) Function

This routine makes a scheme procedure out the C procedure `(*fn)()`. The scheme procedure will require *req* arguments, and accept *opt* optional arguments. The procedure will be documented by the documentation string *doc*. [Note: it is not yet clear how documentation strings will evolve in Guile.]

There are several important considerations to be made when writing the C routine *(*fn)()*.

First of all the C routine has to return type `SCM`.

Second, all arguments passed to the C funcion will be of type `SCM`.

Third: the C routine is now subject to scheme flow control, which means that it could be interrupted at any point, and then reentered. This means that you have to be very careful with operations such as allocating memory, modifying static data ...

Fourth: to get around the latter issue, you can use `GSCM_DEFER_INTS` and `GSCM_ALLOW_INTS`.

GSCM_DEFER_INTS Macro
GSCM_ALLOW_INTS Macro

These macros disable and reenable scheme's flow control. They

6.7 Converting data between C and scheme

Guile provides mechanisms to convert data between C and scheme. This allows new builtin procedures to understand their arguments (which are of type `SCM`) and return values of type `SCM`.

6.7.1 C to scheme

`SCM` **gscm_bool** (`int x`) Function

Returns `#f` if x is zero, `#t` otherwise.

`SCM` **gscm_ulong** (`unsigned long x`) Function
`SCM` **gscm_long** (`long x`) Function
`SCM` **gscm_double** (`double x`) Function
`SCM` **gscm_char** (`char x`) Function
`SCM` **gscm_str** (`char *x, int len`) Function
`SCM` **gscm_str0** (`char *x`) Function

Returns a scheme object with the value of the C quantity *x*.

6.7.2 Scheme to C

`int` **gscm_2_bool** (`SCM obj`) Function
`unsigned long` **gscm_2_ulong** (`SCM obj`) Function
`long` **gscm_2_long** (`SCM obj`) Function
`double` **gscm_2_double** (`SCM obj`) Function
`int` **gscm_2_char** (`SCM obj`) Function
`void` **gscm_2_str** (`char **str_out, int *len_out, SCM *obj`) Function
`void` **gscm_2_str0** (`char **str_out, int *len_out, SCM *obj`) Function

These routines convert the scheme object to the given C type.

Note the distinction between the *str* and *str0*: the former returns with C null-terminated strings; the latter returns a scheme string.

Also note that the string procedures take a pointer to the scheme object *obj*, and that they return the string in a volatile location **str_out*.

6.8 Memory allocation and garbage collection

`SCM` **gscm_mkarray** (`int size`) Function

Allocate memory for a scheme object in a garbage-collector-friendly manner.

6.9 Calling scheme procedures from C

Many of the scheme primitives are available in the `gscm_` interface; they take and return objects of type SCM, and one could basically use them to write C code that mimics scheme code.

I will list these routines here without much explanation, since what they do is the same as documented in section "Standard Procedures" in *R4RS*. But I will point out that when a procedure takes a variable number of arguments (such as `gscm_list`), you should pass the constant *SCM_EOL* from C to signify the end of the list.

`SCM` **gscm_define** (`char *name, SCM val`) Function

Corresponds to the scheme `(define name val)`: it binds a value to the given name (which is a C string).

`SCM` **gscm_cons** (`SCM a, SCM b`) Function
`SCM` **gscm_list** (`SCM l0, SCM l1, ... , GSCM_EOL_MARKER`) Function

These correspond to the scheme `(cons a b)` and `(list l0 l1 ...)` procedures.

`SCM` **gscm_ilength** (`SCM ls`) Function

Returns the length of the list.

`SCM` **gscm_set_car** (`SCM obj, SCM val`) Function
`SCM` **gscm_set_cdr** (`SCM obj, SCM val`) Function

These correspond to the scheme `(set-car! ...)` and `(set-cdr! ...)` procedures.

`SCM` **gscm_car** (`SCM obj`) Function
`SCM` **gscm_cdr** (`SCM obj`) Function

...

`SCM` **gscm_c[ad][ad][ad][ad]r** (`SCM obj`) Function

These correspond to the scheme `(caadar ls)` procedures etc ...

`SCM` **gscm_symbol** (`SCM str, SCM len`) Function
`SCM` **gscm_tmp_symbol** (`SCM str, SCM len`) Function

Takes the given string *str* of length *len* and returns a symbol corresponding to that string.

`SCM` **gscm_vector** (`SCM n, SCM fill`) Function
`SCM` **gscm_vref** (`SCM v, SCM i`) Function
`SCM` **gscm_vset** (`SCM v, SCM i, SCM val`) Function

These correspond to the scheme `(vector n fill)`, `(vref v i)` and `(vset v i value)` procedures.

`SCM` **gscm_make_subr** (`SCM (*fn)(), int req, int opt, int varp, char *doc`) Function
`SCM` **gscm_curry** (`SCM proc, SCM first_arg`) Function

These routines create new scheme procedures; the first form corresponds to `(lambda (...) (...))`; the second curries a procedure by fixing the first argument.

`SCM` **gscm_apply** (`SCM proc, SCM args`) Function

Corresponds to the scheme `(apply proc args ...)`

`SCM` **gscm_catch** (`SCM key, SCM thunk, SCM handler`) Function

`SCM` **gscm_throw** (`SCM key, SCM args`) Function

Corresponds to the scheme `catch` and `throw` procedures, which in Guile are provided as primitives.

`SCM` **gscm_is_eq** (`SCM a, SCM b`) Function

`SCM` **gscm_is_eqv** (`SCM a, SCM b`) Function

`SCM` **gscm_is_equal** (`SCM a, SCM b`) Function

These correspond to the scheme `eq?`, `eqv?` and `equal?` predicates.

`int` **gscm_obj_length** (`SCM obj`) Function

Returns the raw object length.

Concept Index

A

API, -pplet ... 17
-pplet API ... 17
-pplet behaviour ... 18
-pplet data -etrieval ... 17
-pplet pvent Ioop ... 18
-pplet Iibrary ... 16
-pplet Iimitations ... 17
-pplet -equirements ... 16
-pplets ... 16
-pplets, hyperlinks ... 16
-pplets, tnline ... 16
-rguments ... 7

B

bibliography ... 1
browser behaviour ... 18

C

aonfiguring widgets ... 3
aonverting data ... 22
areating widgets ... 2

D

data aonversion ... 22
documentation ... 1
dynamic -oots ... 14

E

prror nessages tn Iibguile ... 20
pvent handlers ... 4
pvents ... 4
pxclusions form nanual ... 1
pxecuting (cheme ... 21
pxtending Guile ... 19
pxtensions mo R4RS ... 1
pxtensions mo (tandard (cheme ... 1

G

ieometry nanagement ... 3
GNU Extension Language ... 1
ioonix ... 7
ioonix pscaping ... 7
ioonix tnitialization ... 7
Green, Anthony ... 14

iscm_ tnterface ... 19
Guile ... 1
Guile d pxtending ... 19
Guile -pplets ... 16
Guile mhreads ... 14

I

o/O pxtensions ... 7
orlam, Gordon ... 16

J

Jaffer, Aubrey ... 19

L

Iibguile ... 19
Iibguile d aonverting data ... 22
Iibguile d data mypes ... 19
Iibguile d prror nessages ... 20
Iibguile d pxecuting (cheme ... 21
Iibguile d headers ... 19
Iibguile d Iinking ... 19
Iibguile d iew erocedures ... 21
Iibguile d -eference nanual ... 19
Iibguile d (tart tnterpreter ... 20
Iibguile tnterface ... 19
Lord, Tom ... 14

M

nime mypes ... 16

N

iew erimitives ... 21
iew erocedures ... 21

P

eacker ... 3
eorts ... 7
eosix ... 7
eosix aalls ... 10
eosix aompliance ... 7
eosix mhreads ... 14
erimitives, iew ... 21
erocedures, iew ... 21

R

-eference nanual ... 1
-eferences ... 1

S

(cheme pxtensions ... 1
SCM data mype ... 19
SCM tnternals ... 19
SCM tnterpreter ... 19
SurfIt! web browser ... 16
(ystem aalls ... 13

T

mcl-lambda ... 2
mhreads ... 14
Tk ... 2
Tk nain Ioop ... 5
Tk nan eages ... 2
Tk lptions ... 2
Tk ererequisites ... 2
Tk widgets -s erocedures ... 3
Tk without Tcl ... 2
mk-main-loop ... 5
mop Ievel ... 20
mutorial ... 1

U

UNIX Iibrary aalls ... 13
UNIX (ystem aalls ... 13
sser nanual ... 1
sser versus erogrammer's nanual ... 1

W

widget aallbacks ... 2
widget aonfiguration ... 3
widget areation ... 2
widget lptions ... 2
widgets -s erocedures ... 3
window nanager ... 4

Procedure Index

This is an alphabetical list of all the procedures and macros in Guile.

%

%access ... 9
%acct ... 13
%chdir ... 8
%chmod ... 8
%chown ... 10
%closedir ... 8
%duplicate-eort ... 8
%execl ... 12
%execlp ... 12
%fdopen ... 9
%fileno ... 9
%fork ... 12
%freopen ... 8
%fseek ... 8
%ftell ... 8
%getcwd ... 8
%getgrgid ... 11
%getgroups ... 10
%getpwuid ... 11
%isatty ... 9
%kill ... 11
%link ... 10
%lstat ... 13
%mkdir ... 8
%mknod ... 13
%nice ... 13
%opendir ... 8
%pipe ... 10
%primitive-nove->fdes ... 9
%putenv ... 10
%readdir ... 8
%readlink ... 13
%redirect-eort ... 8
%rename ... 9
%rmdir ... 8
%select ... 12
%setegid ... 12
%seteuid ... 12
%setgid ... 12
%setuid ... 12
%stat ... 10
%symlink ... 13
%uname ... 12
%utime ... 9
%waitpid ... 11

A

␣fter ... 5
␣pplet-Ioaddata ... 17
␣pplet-Ioadurl ... 17
␣pplet-iewpage ... 17
␣pplet-earsehtml ... 17
␣pplet-window ... 17

B

bell ... 5
bind ... 4
bindtags ... 5
browser-window ... 17
button ... 3

C

aanvas ... 3
aheckbutton ... 3
alipboard ... 5

D

define-␣pplet-merminate ... 16
destroy ... 3

E

pntry ... 3
pnviron ... 13
pxit ... 5

F

fileevent ... 4
focus ... 4
forget ... 4
frame ... 3

G

ietegid ... 12
ieteuid ... 12
ietgid ... 12
ietpid ... 10

ietppid 12
ietuid 12
irab 4
iscm_2_bool 22
iscm_2_ahar 22
iscm_2_double 22
iscm_2_Iong 22
iscm_2_(tr 22
iscm_2_(tr0 22
iscm_2_slong 22
GSCM_ALLOW_oNTS 22
iscm_␣pply 24
iscm_bool 22
iscm_a[ad][ad][ad][ad]r 23
iscm_aar 23
iscm_aatch 24
iscm_adr 23
iscm_ahar 22
iscm_aons 23
iscm_areate_mop_Ievel 20
iscm_aurry 23
GSCM_DEFER_oNTS 22
iscm_define 23
iscm_define_erocedure 21
iscm_destroy_mop_Ievel 20
iscm_double 22
iscm_prror_nsg 20
iscm_pval_file 21
iscm_pval_(tr 21
iscm_tlength 23
iscm_ts_pq 24
iscm_ts_pqual 24
iscm_ts_pqv 24
iscm_Iist 23
iscm_Iong 22
iscm_nake_(ubr 23
iscm_nkarray 23
iscm_lbj_Iength 24
iscm_␣un_(cm 20
iscm_(et_aar 23
iscm_(et_adr 23
iscm_(tr 22
iscm_(tr0 22
iscm_(ymbol 23
iscm_mhreads_tnit_␣ll 20
iscm_mhrow 24
iscm_mmp_(ymbol 23
iscm_slong 22
iscm_vector 23
iscm_vref 23
iscm_vset 23

I

tmage 5
tnfo 4

J

join-mhread 14, 15

L

Iabel 3
Iistbox 3
Iock-nutex 14, 15
Iower 5

M

nake-nutex 14, 15
nenu 3
nenubutton 3
nessage 3

O

lpen-tnput-eipe 10
lpen-lutput-eipe 10
lpen-eipe 10
lption 6

P

eack 3
elace 4

R

␣adiobutton 3
␣aise 6
␣ead-Iine 7
␣ead-Iine! 7
␣ewinddir 8

S

(cale 3
(cm_tnit_itcl 20
(cm_tnit_itk 20
(cm_tnit_toext 20
(cm_tnit_eosix 20
(cm_tnit_snix 20
(crollbar 3
(election 6
(end 6
(etgrent 11

(etpwent 11
(laves 4
(ync 13

T

mext 3
mk 6
mk-nain-Ioop 5
mk_eopup 3
mkwait 5
moplevel 3
mtyname 12

U

smask 9
snlock-nutex 14, 15
spdate 3, 5

W

WIDGET 3
winfo 6
with-iew-mhread 14, 15
wm 4
write-Iine 8

Y

bield 14, 15

Types - Variables - Constants

This is an alphabetical list of all the important data types, variables and constants defined in the Guile Programmer's Manual. Note: you should also look at the indices in the Guile User Manual.

'

'applet 16

A

␣pplet-pmbedindex 17
␣pplet-window-iame 17

B

browser-window-iame 17

G

GSCM_ERROR_OPENING_FILE 21
GSCM_ERROR_OPENING_oNIT_FILE 21
GSCM_oLLEGALLY_REENTERED 21
GSCM_OK 19, 21
GSCM_OUT_OF_MEM 21
GSCM_QUIT 21
GSCM_RESTART 21
GSCM_(tatus 19
GSCM_mop_level 19

S

SCM 19

Table of Contents

1 What goes in this manual 1

1.1 What does NOT go in this manual 1

2 Guile Tk reference 2

2.1 Tk preliminaries 2
2.2 Creating widgets 2
2.3 Widgets as procedures 3
2.4 Geometry management procedures 3
2.5 Event handling procedures 4
2.6 Miscellaneous Tk procedures 5

3 Goonix reference 7

3.1 Goonix preliminaries 7
3.2 I/O extensions to standard scheme 7
3.3 Posix system and library calls 10
3.4 UNIX system and library calls 13

4 Guile threads reference 14

4.1 Low level thread primitives 14
4.2 Higher level thread procedures 15

5 Guile applet specification 16

5.1 HTML applet extension 16
5.2 Applet requirements 16
5.3 Applet API 17
5.4 Applet behavior 18
5.5 Browser behavior 18

6 Libguile reference 19

6.1 Libguile preliminaries 19
6.2 Data types defined by libguile 19
6.3 Starting and controlling the interpreter 20
6.4 Error messages 20
6.5 Executing scheme code 21
6.6 Defining new scheme procedures in C 21
6.7 Converting data between C and scheme 22
6.7.1 C to scheme 22
6.7.2 Scheme to C 22
6.8 Memory allocation and garbage collection 23
6.9 Calling scheme procedures from C 23

Concept Index 25

Procedure Index **27**

Types - Variables - Constants **30**